Shared Library Injection in Android

By Shunix

Published: 2016-03-22 · Archived: 2026-04-02 12:39:58 UTC

March 22, 2016

Introduction

Injection is a technique that enable us running our code inside a remote process. Usually, we compile the code into a shared library and force the remote process to load it, so we could execute our code. We call this trick "shared library injection".

If you're familiar with Linux, you probably heard of the **ptrace** system call. This system call can be found in most Unix-like systems. By using ptrace, we can get control of the target process and manipulate the internal details of it. It's used by the well-known debugger **gdb** to trace processes.

Android is based on Linux. According to what I wrote above, the ptrace call can be found in Android as well. We need this system call to achieve the injection.

Ptrace

I took a piece from the Linux man page of ptrace.

> The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

Synopsis of ptrace:

```
 #include <sys/ptrace.h>
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

Tracer process first need to call ptrace with PTACE_ATTACH as the first argument. If the call succeed, we've gained the control of the tracee process. Then we can send command to the tracee process by changing the first argument. The first argument *request* is a enum, it has many values, we only need some of them to achieve our injection. I'll list these values below and explain the meaning of them roughly.

- PTRACE_ATTACH
  Attach to the process specified in pid.
- PTRACE_PEEKTEXT, PTRACE_PEEKDATA
  Read a word from memory of the tracee process.

- PTRACE_POKETEXT, PTRACE_POKEDATA
  Copy a word to the tracee's memory.
- PTRACE_GETREGS
  Get the general-purpose registers of tracee process.
- PTRACE_SETREGS
  Modify the general-purpose registers of tracee process.
- PTRACE_CONT
  Resume the stopped tracee process.
- PTRACE_DETACH
  Detach from the tracee process.

ARM Call Convention

Before we get started with the injection, we need to know the call convention in ARM architecture.
When calling a function on ARM, the first 4 arguments is stored in general-purpose registers **R0-R3**. If the function has more than 4 arguments, the remainder is pushed into the stack and the **SP** register is changed accordingly. Then the function address will be put into **PC** register and the return address in **LR** register. When the function finished execution, the return value will reside in **R0** register. It's pretty simple, right?

Brief of Injection

Now we're done with the prior knowledge. Based on what I've talked above, to achieve the injection, we need to do the following things:

- Attach to the target process, making it a tracee of our process.
- Load shared library to the tracee process.
- Get the address of our function.
- Save the current registers of tracee process.
- Pass the arguments by ARM call convention.
- Set **LR** to 0, so we can catch the **SIGSEGV** after the call.
- Set **PC** to the address of our function.
- Mask **PC** and **CPSR** according to the mode (thumb or arm).
- Resume the tracee process and wait for the **SIGSEGV** signal.
- Get the return value in **R0**.
- Restore the registers of tracee process.

It seems very easy, but now we have a problem here. How can we load our shared library into the tracee process? It's the core of the injection, I'll talk about it in detail in the next section.

Implementation

Generally speaking, there're two ways to load shared library into remote process.
The first one require the knowledge of ARM assembly. First we allocate a piece of memory in the remote process, then we write the necessary parameters needed by **dlopen** system call and **shellcode** to this memory region.

Finally we execute the **shellcode**, all work is done. The **shellcode** is a small piece of code written in ARM assembly that capable of loading the shared library. While the assembly is hard to read, so I'm not gonna take this method.
The second way to load shared library is much easier. As we haven known the ARM call convention, we can write a util function to call the functions in remote process.
The code of util function:

```
long CallRemoteFunction(pid_t pid, long function_addr, long* args, size_t argc) {
  struct pt_regs regs;
  // backup the original regs
  struct pt_regs backup_regs;
  ptrace(PTRACE_GETREGS, pid, NULL, &regs);
  memcpy(&backup_regs, &regs, sizeof(struct pt_regs));
  // put the first 4 args to r0-r3
  for(int i = 0; i < argc && i < 4; ++i) {
    regs.uregs[i] = args[i];
  }
  // push the remainder to stack
  if (argc > 4) {
    regs.ARM_sp -= (argc - 4) * sizeof(long);
    long* data = args + 4;
    PtraceWrite(pid, (uint8_t*)regs.ARM_sp, (uint8_t*)data, (argc - 4) * sizeof(long));
  }
  // set return addr to 0, so we could catch SIGSEGV
  regs.ARM_lr = 0;
  regs.ARM_pc = function_addr;
  if (regs.ARM_pc & 1) {
    // thumb
    regs.ARM_pc &= (~1u);
    regs.ARM_cpsr |= CPSR_T_MASK;
  } else {
    // arm
    regs.ARM_cpsr &= ~CPSR_T_MASK;
  }
  ptrace(PTRACE_SETREGS, pid, NULL, &regs);
  ptrace(PTRACE_CONT, pid, NULL, NULL);
  waitpid(pid, NULL, WUNTRACED);
  // to get return value;
  ptrace(PTRACE_GETREGS, pid, NULL, &regs);
  ptrace(PTRACE_SETREGS, pid, NULL, &backup_regs);
  // Fuction return value
  return regs.ARM_r0;
}
```

With the util function, task is simpler. I did the following things:

1. Get the **mmap**, **dlopen**, **dlsym**, **dlclose** fuction address in remote process.
2. Call **mmap** in the remote process to allocate a memory region.
3. Write the shared library path to the memory region.
4. Load shared library by calling **dlopen** in remote process.
5. Get our custom function address using **dlsym**.
6. Call custom fuction.
7. Unload the shared library using **dlclose**.

But the work is not done yet. There is a problem with the first step due to address space layout randomization.

Address Space Layout Randomization

In order to prevent the buffer overflow attacks, most operating systems provided a technique named "Address Space Layout Randomization", commonly called **ASLR**. With this technique, the start position of key data area is random, including the position of stack and heap, etc.
As for Android, position-independent executable support was added in Android 4.1. Android 5.0 dropped non-PIE support and requires all dynamically linked binaries to be position independent. Thus the function address is not the same between processes. So we need a way to bypass the ASLR and get the target function address in remote process.
Look at what we have(take **mmap** as example):

- **mmap** address in our own process.
- base address of **/system/lib/libc.so** (where **mmap** lies in) in our process
- base address of **/system/lib/libc.so** (where **mmap** lies in) in remote process

We know that the offset of **mmap** from the library base address is certain. Based on the three address above, we can get the **mmap** address in remote process now.

Remote mmap address = local mmap address - local library base address + remote library base address.

We can get other function address in the same way, thus we've successfully bypassed the ASLR.

At this point, we've achieved the shared library injection. The code of this post can be found on my Github repo "TinyInjector". (This repo is private currently, I'll open source this repo at the right time). I have test the code by injecting WeChat and Chrome on Android 4.4.4.
